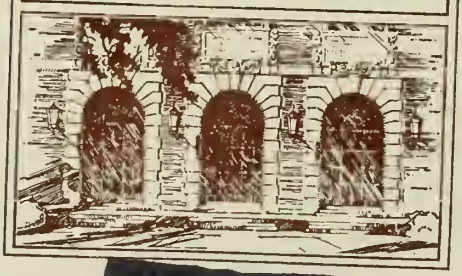
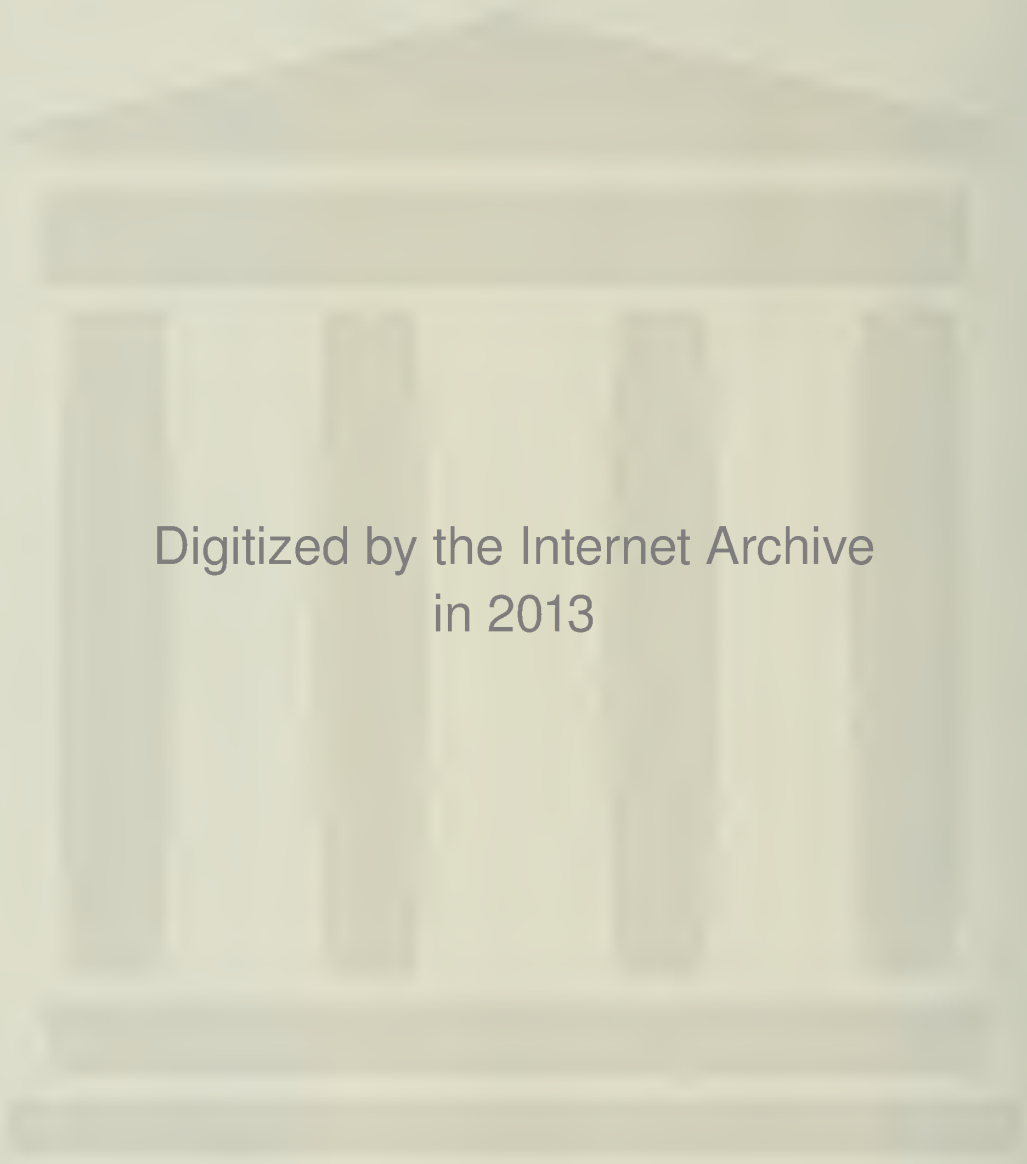




UIUCDCS-R-76-819

THE DESIGN OF WITS

A STUDENT COMPILER SYSTEM ON PLATO IV

by

Lawrence Allen White

July, 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE DESIGN OF WITS

A Student Compiler System on Plato IV

BY

LAWRENCE ALLAN WHITE

E.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

TABLE OF CONTENTS

PREFACE

This thesis discusses the design of an interactive student compiler system called WITS, an acronym for "White's Interactive Timesharing System." The WITS system currently contains two editors, two filing lessons, and two compilers. Work is in progress on creating additional editors and compilers, in order to better meet the needs of its users. These compilers have been implemented on the Plato IV computer system, and are being used by students. Included in this thesis is a description of the Plato IV system and how it affected the development of WITS.

Chapter 1

Introduction

The Plato IV system [Alpert,1970] is a computer based educational system developed at the University of Illinois. Various user groups at Plato have been interested in designing or using interactive compilers for student programs. Two such systems of compilers have been implemented on Plato. This thesis discusses the design considerations that went into the implementation of one of these compiler systems and the results of that design. The compiler system described here is called WITS, an acronym for "White's Interactive Timesharing System." Throughout this thesis, occasional comparisons are made to the other compiler system, CAPS [Wilcox,1976], which has been implemented by the Computer Science Department at the University of Illinois.

In order to give some basis for the discussion of the design chosen for WITS, the first two chapters describe the users of the WITS compilers and the computer system on which WITS was implemented. Chapter 4 describes the goals the WITS designers had in implementing their system. Chapters 5 and 6 describe WITS as it currently is, and how it became that way.

Chapter 2

Users of Interactive Compilers on Plato

Before the implementation of any computer programming system, consideration must be given to the type of users the system will have, and what their needs are. In designing WITS, several user groups were kept in mind, in an attempt to make the system general enough to be used by many groups. The user groups the author has had experience with are described below, along with their needs as seen by the compiler designers.

2.1. Educational Psychology

BASIC [Kemeny,1967] is a simple programming language, primarily intended for numerical applications. Because of its simplicity and its acceptance in the education environment, it was chosen as the first language to be implemented on Plato. The compiler was written for Professor Kenneth Travers of the Secondary Education Department, and for Axel T. Schreiner of the Mathematics Department, both at the University of Illinois. As understood by the compiler designers, Dr. Travers was instructing future teachers in how to teach BASIC. Since his students would later, in most cases, be using small interactive systems utilizing Teletypes for communication, Dr. Travers wanted the user interface of the BASIC system we built on Plato to resemble such a system. In particular, his requirements were

as follows:

1) User interface similar to standard interactive system.
   A) Teletype like appearance.
   B) Clear distinction between editing and compiling.
   C) Command directives, rather than user choice page.

2) Provide simple program preparation facilities.
   A) Permanent program storage space for 10 users.
   B) Line-oriented text editor.

3) Compile and execute small BASIC programs.

Unfortunately for the Education Department's students, the Plato system was still experimental at that time, and rather unstable. Additionally, these students were the first to use the forerunner of the WITS system, and they found many bugs in the editor, filing system, and BASIC interpreter. To their credit, they stuck with the system for two or three semesters while the Plato system was stabilizing and we were fixing problems in our lessons.

2.2. Computer Calculus

One of the co-authors of that first system, and the real designer of our BASIC compiler, Axel T. Schreiner, was at that time designing "Computer Calculus" [Schreiner,1972], a laboratory course supplementing regular Calculus instruction at the University of Illinois. His students were writing small numerical analysis and graphical programs using CALL/360-OS BASIC [IBM] on PLORTS [Gear,1969], a local interactive timesharing system. For the purpose of Computer Calculus, the Mathematics Department had set up a laboratory room connected to

the IBM/360-75 operated by the Computing Services Office at the University of Illinois. This laboratory had 16 teletypes for program preparation and alphanumeric input and output, along with a small Hewlett-Packard flat-bed plotter, connected to one of the teletypes, for graphical output.

Mr. Schreiner became interested in using Plato for the graphics portion of the Computer Calculus classes, and wrote some instructional lessons on Plato for this. Later he expanded his interests to include using Plato for a section of his students to run BASIC programs. For Computer Calculus students to use Plato for BASIC, they would need essentially the same facilities as Professor Travers needed, but for 20 students simultaneously, not 10.

To meet the goals of these two user groups, Educational Psychology and Computer Calculus, the user interface of the first BASIC system implemented on Plato looked very much like PLORTS, to the extent that some of the editing commands were identical to those on PLORTS. This system was used by Education and Calculus students until a more advanced system, called WITS, was implemented.

2.3. University High School

The heart of the WITS system was written in the fall of 1973, using some new features of the Plato system, and was first used the following spring. WITS allowed for the connection of any number of user groups to one filing system, all using the

same editor and compiler. The first new group of users was from University High School, which already had access to Plato, and whose instructors were interested in teaching their students to program.

The "Unihi" students had no previous experience with other computer systems, and in many cases, WITS was their first encounter with Plato. These students, rather than using programming to learn another subject, as were the Calculus students, were learning about computers, computer systems, and programming. As such, the WITS system was one of the things they were learning about, rather than being just a tool in the study of something else.

The basic requirements for the University High School students were slightly different from those of the other user groups previously described. What they needed WITS to provide are described below:

1) Provide a programming system at small additional cost to the school.

2) Be simple enough to allow some quick achievements for students, powerful enough to hold their interest for a semester or two.

3) Be simple enough to explore by trial and error, safe enough so students can not accidently hurt anything.

4) Provide filing space for 35 students.

The original WITS system filled all these needs, so the University High School students started using it. As WITS has expanded to include a Fortran compiler and another editor and

filing system, Unihi students have continued to utilize all the capabilities of the WITS system.

2.4. CS 105

The Computer Science Department at the University of Illinois has been working for a few years to automate their introductory programming courses [Nievergelt,1974]. To do this, members of the department have written many instructional lessons on Plato, and have implemented interactive diagnostic compilers for subsets of several programming languages [Wilcox,1976]. Of particular interest is the CS105 programming course, with about 1000 students each semester. The WITS designers have tried to expand their compiler system to support this group, but due to the large number of students, this attempt has been only partially successful.

CS105 is an introductory programming course for business students. In many cases, this will be their only programming course, which they take because it is required of them. Such a group is, on the whole, only interested in getting their machine problems done with the least amount of effort. They are not interested in learning how editors or filing systems work (unless that is assigned to them), and they are very intolerant when they can not get the system to work.

Since there are so many of these students, they are scheduled rather tightly, and are unlikely to come in on their own time, unless it is to finish a late programming assignment.

Due to the large number of students, a different organization of resources may be required in WITS to handle them all. Below are listed the major requirements to use WITS for Introductory Computer Science Courses at the University of Illinois:

1) Provide Fortran and/or PL/1 compilers.
2) Support 40 simultaneous users.
3) Provide filing space for 1000 students.
4) Use a minimum of costly system resources.
5) Provide self-explanatory editors and filing systems.

Expected use of WITS in the near future by the CS department appears rather minimal at this time. This is due to the department having its own compilers on Plato, and to the availability of batch compilers on CSC's IBM System 360/75. One way WITS might be used is as an intermediate step between their Plato compilers and the 360. The students could learn the basic features of a programming language using the department's diagnostic compilers on Plato, and could then move to WITS for writing longer programs, and later move to the 360 to learn about a batch oriented system, printed output, file I/O, and still larger programs.

2.5. Visitors

In addition to serving the organized groups described above, the WITS designers have encouraged people to set up Plato records so students might "visit" the WITS system. Several such

visitor records have been created for students to practice programming on their own. In some cases, these students have even been provided with disk space for storing their programs between sessions. If they do have such space, it is shared between all users of their Plato signon. For WITS to support such visitors, the following are needed:

1) Visitors may write and run programs even if they are not allocated file space.

2) They should not be able to interfere with regular users of WITS.

3) Help pages should be provided to answer their questions without needing an instructional book or teacher.

4) WITS should work reasonably well with a single user.

2.6. Future Users

In the next few years it is expected that the Plato system will expand to include several computers at different universities. The designers of WITS are trying to make their system transportable to other Plato systems. To do so, they expect the following:

1) Diverse user needs, as above.

2) Visitors may be familiar with the Plato editor, and want a similar editor for WITS.

3) Resource use by WITS may be a critical factor.

4) WITS data structures should be stable, even if the host system is occasionally unstable, since the WITS designers will not be around to correct problems.

Chapter 3

The Plato IV System

When designing a compiler system for a particular environment, it is often just as important to understand what that environment can support as it is to understand what the users will need. In early 1973 it was thought, for one reason or another, that Plato couldn't support student compilers. Claims were made that Tutor, Plato's programming language, was not powerful enough for a compiler; Tutor's data areas and structures were too small or too simple; or Plato's CPU capacity was insufficient. That was before Axel Schriner designed, and he and the author implemented, their first BASIC system. They showed that Plato could support a compiler system, if that system were designed and implemented carefully, and if the users were running "reasonable" programs. Since the capabilities of Plato strongly affected the design of WITS, the next few sections describe Plato, so that later discussions of WITS will be more firmly based.

The goal in the design of Plato was to provide high quality education at low cost [Bitzer, 1973]. The method of achieving this goal was to use a powerful computer to enable the creation of quality instructional materials, and to keep the cost per student low by running hundreds of students. One of the important characteristics of quality CAI is fast and flexible

feedback to the student, giving better reinforcement in learning. How Plato provides fast response to student input is described in the next two sections. One section is concerned with the central computer, the other section outlines the communications system.

3.1. Central Computer Architecture

The computer on which Plato runs is one of Control Data Corporation's 6000 series. Specifically at the Urbana Plato system, a Cyber 73 is used. The Cyber 73 has two Central Processing Units (CPUs), and ten Peripheral Processing Units (PPUs). The PPUs control communications between the computer and other devices, such as printers, tape drives, and the disk system. In addition, the PPUs handle the main part of system job scheduling, telling the CPUs what to do. The CPUs do most of the work in running programs, including the execution of Plato lessons. More on the design of the central computer may be found in [Thorton,1970], though he basically discusses the CDC 6600.

Many standard interactive computer systems employ something known as "swapping" in their operation. Swapping involves keeping the information needed for all users on some secondary memory device, generally slower and cheaper than main memory, and transfering one user's information into main memory whenever he needs to execute. This is mainly an economic feature, allowing the computer to get by with less main memory than would

be needed if all active users were kept there all the time. The transfer speed between the secondary memory and main memory is often an important factor in how well, or how many users, the system can run, both of which are important aspects of the Plato system.

In order to provide good response time for many users, Plato uses Control Data Corporation's Extended Core Storage (ECS) for its secondary memory storage. The transfer rate between ECS and Plato's main memory (called Central Memory or CM) is 600 million bits per second. This high transfer rate allows Plato to swap its users in and out of CM very quickly. However, in addition to Plato using ECS to perform a fast swap of users, Plato's programming language allows users to directly access data areas in ECS. Thus, ECS goes beyond just being a transparent system implementation feature, invisible to the users; it affects the whole design of data manipulation programs, including those in the WITS system. The data areas in ECS that a lesson may reference are described in Section 3.5.

Many computer systems use disk packs for their swapping medium. Due to its much longer access time and lower transfer rate, disk packs are used by Plato only for the permanent storage of infrequently accessed data. Among the types of information Plato stores on disk are the permanent records for all users, the source code for lessons, files for collection of data on active students, and files for normal lessons to store data in. At the start of a session, a user's information is

read from disk into ECS, and it is stored back on disk when he signs off. By storing a user's information in ECS during a session, the number of disk accesses required to run a student is reduced, and the time needed to access a user's information is shortened. A similar strategy is employed with most other data references. For example, when editing the source code of Plato lessons, the text is brought into ECS in blocks of a few hundred words, from where it is edited and later returned to disk.

3.2. Terminal Communications

In providing fast response to user input, two things are important, the time required to set up a user for processing and the communication time to the terminal. The swapping of users was briefly discussed in the preceding section, while this section describes the communication to the terminal. This communication did not, however, affect the design of WITS as much as did Plato's architecture, so it is described even more briefly. More complete information is available from [Stifle,1972].

Each Plato terminal is designed to connect to a voice grade phone line (1260 baud), which is in turn connected to a Plato Site Controller. Each site controller handles the input and output of 32 terminals. Input to the computer from each site controller is provided by a phone line to the Network Interface Unit or NIU [Tucker,1971]. Output from the central computer is

passed to the NIU, where it is formatted into a video signal. Output to 1008 terminals reaches over a video cable from the NIU to all site controllers, each of which picks off the output meant for their own 32 terminals.

At the central computer site, input and output through a channel to the NIU is directed by a program running on a PPU. This PPU program stores key input from the terminals in ECS and tells a CPU program where to find the key input. The PPU program is also responsible for picking up the output waiting to go to the terminals and sending it to the NIU. The Plato interpreter polls the input tables looking for keys every 50 milliseconds, giving each terminal with a key a timeslice of computation. In addition to keys from the NIU, Plato may generate pseudo-keys to indicate that a terminal needs to continue some computation that was previously interrupted.

3.3. Student Terminal

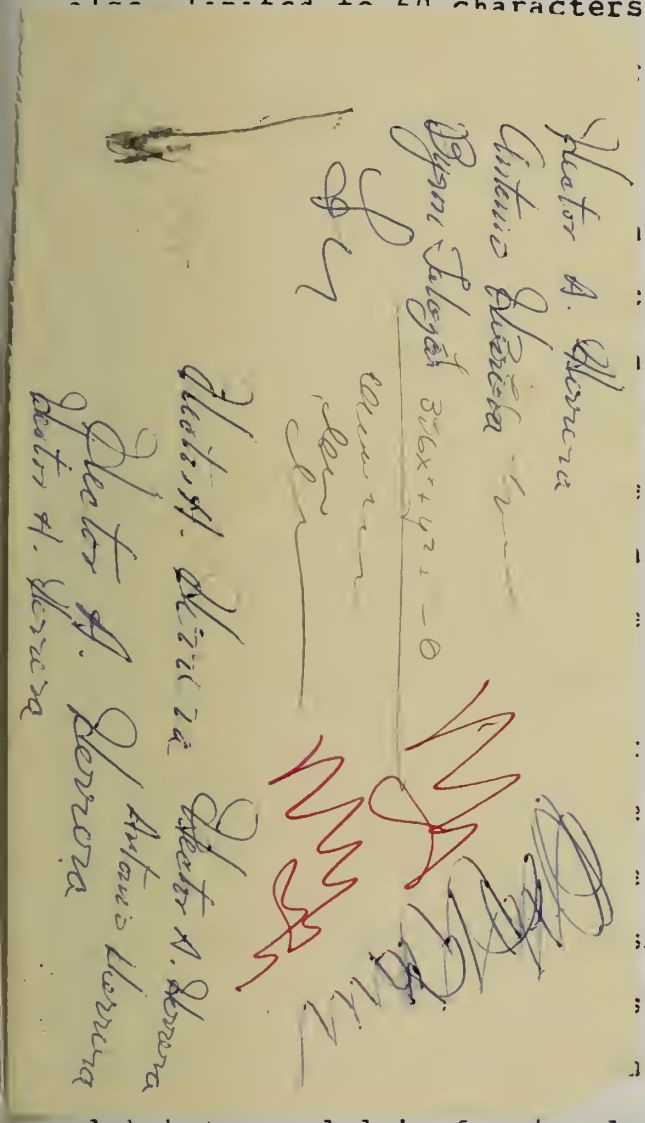
The Plato IV student terminal is designed to provide features desireable for many teaching applications. The basic terminal includes a graphic display panel for output and a keyset for input. Optional accessories include a slide selector for displaying slides on the panel, an audio device supporting recorded messages, and a touch panel for providing touch sensitive input. Jacks are provided on the back of the terminal for adding external devices. The bandwidth to the terminal is 1260 baud, higher than that used at most timesharing systems.

The terminal features that affected the development of WITS are described below. A more complete terminal description may be obtained from [Stifle,1973].

The display device in the terminal is an 8 1/2 inch square plasma pannel, containing a grid of 512 by 512 dots, each of which may be illuminated independently. The plasma panel display needs no refreshing to hold a display, as does a CRT, and its orange dots on a black background provide high contrast, flicker free output. The display may be referenced in "fine grid" notation, using (x,y) pairs in the range from (0,0) to (511,511) to indicate points. Alternately, the panel may be referenced in "coarse grid" notation, as 32 lines of 64 characters each. Each character position is 16 dots high by 8 dots wide.

Since student programs in the WITS system may be longer than 32 lines, some method is required to show only part of a student's program at once. In a "teletype" editor for BASIC programs, this is done by writing lines down the screen and then wrapping around to the top of the screen. In a "cursor" editor, the student's workspace is divided into three "pages" of 30 lines each, with the student being able to move from one page to another. In a "tutor" editor, so called because of its resemblance to the standard Plato source editor, the student moves a "frame" around within his program, and all lines within the frame are displayed on his screen.

Since the display width is only 64 characters, the maximum length of a line of text in WITS was chosen as 60 characters, to leave space for displaying line numbers. Continuation lines did not appear to be necessary, but were implemented in the Fortran compiler as an extra feature. Output from a running program is also limited to 60 characters, a significant departure from the width of 120 or more characters



the plasma panel is accomplished ter generator. This generator which are stored the dot patterns Each memory contains the rs, each character consisting of 8 l can take from the character rn that represents a character and screen. Two of these memories are acters, i.e., the representations considered to be the most useful. loaded under program control with ires. This "alternate" character standard uses include loading a few mathematics lessons, or whole alphabets needed in foreign language lessons. Another very nice way to use the alternate character set is to design pictures using several characters from the alternate character set, and to display the picture very rapidly, at character display speed, rather than at line drawing speed. A good example of this is

The terminal features that affected the development of WITS are described below. A more complete terminal description may be obtained from [Stifle,1973].

The display device in the terminal is an 8 1/2 inch square plasma pannel, containing a grid of 512 by 512 dots. each of which may be illuminated independen
display needs no refreshing to hold
and its orange dots on a black backgrou
flicker free cutput. The display may
grid" notation, using (x,y) pairs i
(511,511) to indicate points. Alternat
referenced in "ccarse grid" notatic
characters each. Each character positi
dots wide.

Since student programs in the
than 32 lines, some method is required
student's program at once. In a "t
programs, this is done by writing lines
wrapping around to the top of the scree
the student's workspace is divided int
lines each, with the student being able
another. In a "tutor" editor, so called because of its resemblance to the standard Plato source editor, the student moves a "frame" around within his program, and all lines within the frame are displayed on his screen.

Since the display width is only 64 characters, the maximum length of a line of text in WITS was chosen as 60 characters, to leave space for displaying line numbers. Continuation lines did not appear to be necessary, but were implemented in the Fortran compiler as an extra feature. Output from a running program is also limited to 60 characters, a significant departure from the more conventional line printer width of 120 or more characters found on most systems.

Alphanumeric writing on the plasma panel is accomplished using the terminal's character generator. This generator implcys four "memories," in which are stored the dot patterns that represent characters. Each memory contains the representation of 63 characters, each character consisting of 8 times 16 dots. The terminal can take from the character memories the 8 by 16 bit pattern that represents a character and display it anywhere on the screen. Two of these memories are built with the "standard" characters, i.e., the representations of the 126 characters that are considered to be the most useful. The other two memories may be loaded under program control with any character set the user desires. This "alternate" character set has many uses. Some standard uses include loading a few special characters needed in mathematics lessons, or whole alphabets needed in foreign language lessons. Another very nice way to use the alternate character set is to design pictures using several characters from the alternate character set, and to display the picture very rapidly, at character display speed, rather than at line drawing speed. A good example of this is

the Plato IV version of the biology lesson "fly" [Hyatt,1972], in which fruit flys with various genetic traits are displayed using combinations of alternate characters.

One way of thinking of the loadable character set is as a way of changing the display form of Plato internal character codes. Conversely, if the loadable character set contains a rearrangement of the standard character set, this can be thought of as rearranging the internal codes used to store characters. Standard Plato format for character storage is a mixture of six, 12, and 18 bit codes. Upper case letters, such as commonly used by programming systems, are 12 bit codes in standard Plato format. The lessons in the WITS system use the alternate character set to rearrange the internal codes of characters, so each supported character can be stored in a six bit code. This decreases the space needed to store a program, and also, decreases the complexity of the compilers and editors in the WITS system.

The terminal contains a vector generator, allowing it to draw lines, given only the endpoints of each line. This allowed the incorporation of some plotting features in the implemented BASIC language. On PLORTS, one flat bed plotter was shared between all students of each Computer Calculus section. On Plato, graphic plotting was available at each student terminal.

3.4. Software

Plato supports but one language, Tutor [Sherwood,1974], for its users. Tutor has developed under the direction of Paul J. Tenczar, head of Plato's system software staff, as a command oriented, second generation language. It is much more powerful and easy to use in teaching applications than most programming languages and many CAI languages [Lower,1976]. In non-teaching applications, such as the WITS system, Tutor provides as much power as many general programming languages, though it is sometimes awkward in such situations. To use Tutor for non-teaching applications often requires much training and practice, after which it may still be difficult to use.

A Tutor program is called a "lesson," and consists of a few hundred to a few thousand lines of code. Each lesson is further divided into "units," each containing anywhere from one to a few hundred lines of code. Program sequencing has traditionally been from unit to unit, with each unit performing a specific function. This allows easy structuring of a lesson into functional modules.

Each Tutor command may require many machine operations, and for the Tutor compiler to generate machine code for each command would result in very large binaries for lessons. To reduce the code needed in each lesson, the compiler "condenses" Tutor code into an easily interpretable form for the runtime system to interpret. To save execution time, however, the compiler generates machine code for expressions, which may then be

directly executed by the computer.

Like all the other information needed for each user, lessons are kept in ECS and swapped into CM only when needed. The organization of a Tutor lesson into units relates itself very well with Plato's architecture. Each unit is a self contained part of a lesson, and gets swapped into CM independently of the rest of the lesson. In trying to write fast Tutor code, one remembers that there is some overhead in bringing units into CM, and that it may be faster to branch within the unit than to load another unit from ECS.

The data referenced by Tutor lessons is kept separate from the lesson itself, and the interpretable code within the lesson is never changed during execution. Thus, the condensed code for all lessons is automatically reentrant and reuseable as far as Plato is concerned. This allows a minor speedup by not having to write units back to ECS after execution, and a major savings in ECS use by allowing any number of users to share the ECS binary of a lesson.

3.5. Tutor Data Areas

Previous sections have mentioned the data storage resources the Plato system has, namely CM, ECS, and disk space. In the distant past, normal Tutor lessons could only access data in Central Memory, with Plato controlling all reading and writing of ECS. Since then, additional commands have been added to Tutor to allow lessons to access data in CM, ECS, and on disk.

Since lessons in the WITS system use all three of these storage areas, their characteristics and use are described below.

Each user, when he signs onto the Plato system, is allocated a 400-500 word area of ECS, called the "student bank." A portion of this, 150 words long, is called "student variables," and can be modified explicitly by the code of whatever lesson the user is currently in. The rest of the student bank is used by Plato to store status information needed during a session. Much of the student bank, including the student variables, is saved on disk between sessions, often allowing the student to resume a lesson at a later session at the point from where he left it earlier.

In addition to student variables, a Tutor lesson may request an additional area in ECS called "common." Whereas each user has his own student variables, there is only one copy of each common, shared between all users of a lesson. A common may reside permanently on disk when not in use by any lesson, being brought into ECS by Plato when needed, and written to disk when no longer referenced; or it may be "temporary" common, which is not saved on disk between sessions. In addition, a number of lessons may share the same ECS copy of a permanent common. A common may now be up to 8050 words long, and is ideal for the storage of small to medium amounts of data, due to its low overhead in disk accesses and its easy accessibility in ECS.

If a Tutor lesson needs more storage space, it can request an additional area of ECS that will be individual to the user. This area is called "storage," and may be up to 1500 words long. Unlike student variables or common, Plato does not save storage variables on disk between sessions. Like student variables, each user of a lesson will have his own set of storage variables. Storage is not shared between users, as is common.

All three of these memory storage areas, student variables, common, and storage, reside in ECS. To perform calculations on the data in these areas, however, they must be in Central Memory. Some method is needed for Tutor lessons to move this data back and forth between ECS and CM. Plato will, at the start of every timeslice, load the student bank, including the student variables, into a fixed buffer in CM. The student variables may then be accessed by the lesson, and they are written back to ECS by Plato at the end of the timeslice. Since Plato automatically does this for the user, few people distinguish between the ECS copy of the student variables and the CM copy of the student variables.

An area is also allocated in CM for the loading and unloading of common and storage. The area where student variables are loaded is called *n* variables, while the area for common and storage loading is called *nc* variables. Since *nc* variables are only 1500 words long, only portions of common and storage can be loaded at one time. Commands are provided in Tutor to direct Plato to automatically load and unload these

areas of common and storage into *nc* variables at the start and end of every timeslice. It is up to the lesson to decide what to load, and to see that the loaded areas do not overlap. During the timeslice, the lesson can do any calculations it wants on the *nc* variables, and those *nc* variables that are "protected" by being loaded from ECS will be saved in ECS until the next timeslice. The *nc* variables that were not "protected" will be zero on entry to the next timeslice. It has been possible in the past to reference unprotected *nc* variables during a timeslice, which provided very convenient "temporary variables," whose values disappeared at the end of the timeslice. This is no longer safe, since users no longer know when their timeslice will end.

Until recently, Tutor lessons could not access disk directly. Their only areas of permanent storage were in a user's student variables, which Plato would save on disk when the user signed off, or in permanent common, which Plato would return to disk when the last user left the last lesson referencing that common. Direct disk accessing was provided for Tutor lessons in order to allow users to store large amounts of infrequently accessed data, such as is needed in a filing system for student programs.

Disk files that Tutor lessons may access are called "datasets," and are organized into fixed size records which are anywhere from 64 to 512 words long. Information read from disk may be sent to the ECS copies of student variables, common, or

more traditionally, storage. Users of one lesson may be connected to different datasets, and a user may switch from one dataset to another during execution.

3.6. Resource Limitations

When the designers of the WITS system first started on Plato, the system was rather small compared to its current size. At that time it had only a single CDC 6400 CPU, only 500K words of ECS, and a new 841 disk system with three or four disk packs. At present, the Plato system is attempting to expand from two 6400 CPUs in one main frame to four CPUs in two main frames. Their disk system is now both larger and faster, and consists of about a dozen 844 disk drives. ECS has been expanded to the maximum of two million words. Also during the past few years, the number of users on Plato has increased tremendously. In order to assure a minimum standard of service to all users, the Plato staff has placed some restrictions on all scarce resources. Naturally, not all users are expected to use all resources to the limits.

The system limits on resource use, particularly on processing power, ECS use, and disk accesses, have been outlined by the Plato system staff. The following values are taken from [Tenczar,1974], where the expected limits for the following two or three years were described. Enough processing power is available to allow two thousand Central Processor instructions per user per second. ECS space is available for about 1500

words per terminal, and disk accesses are available for about one access per user per minute. These limits are what the system has available, but much of each resource are used by the system in just running the student. The conclusions reached were that Platc could support the following type of operation: "four terminals sharing a 6000 word lesson for at least 15 minutes, using 2000-3000 instructions per terminal per second."

Chapter 4

WITS Design Philosophy

In previcus chapters, we have discussed two of the influences cn the design of WITS, the expected users of WITS and the ccmputer system on which it would run. This chapter discusses our design philosophy, that is, not what we did, but our attitude as we did it. Our first goal in designing the WITS system was tc provide an interactive compiler system that many people could use. Our second goal was to determine how far we ccould stretch the capabilities of the Plato system, to find out what it could really support. In this second respect, we have been surprised: Platc has been able to support more than we had oriqinally expected.

4.1. Ccmplete System

Since cne of our goals was to serve as many users as pcssible, we had to provide enough features so that they would want tc use WITS. For those people who were already using an established timesharing system, such as PLORTS, we had to provide as many features of the existing system as possible. Fcr those who were not already using a timesharing system, it was impcrtant that cur system be easy to use even if one had no previous experience. In discussing a "complete" system, we mean a logical structure of filing systems, editors, compilers,

interpreters, and debug packages. Each of these components is described below.

The purpose of an editor within the WITS system is to allow the student to write and modify a program. It should be organized in a manner to make program preparation a simple, easy to understand operation. If a single editor is not suitable for editing programs in different languages simply, multiple editors are needed, each one suited for a different programming language. If, in addition, the users of the editors cover a wide range of sophistication and experience, it may be possible to provide more complex, more powerful editors for those with greater ability, while still filling the needs of more inexperienced users with the simpler editors.

The purpose of a filing lesson within the WITS system is, as with the source editors, to facilitate program preparation. A simple filing system will allow the user to save a program and retrieve it later. More complex filing systems may allow a two level file structure, manipulations of multiple files, access of files by multiple users, or any number of other options. Two important features of filing systems are the resources used for saving files and the stability of the filing structure under adverse conditions.

Since Plato only supports a single language for its users, for students to write programs in another language requires the construction of a compiler for that language and an interpreter for the internal code generated by that compiler. The job of

the compiler is to perform syntactic and semantic analysis of the user's program while generating an easily interpretable internal form of his program. An interpreter then runs over this internal form executing each statement. Upon termination of the running program, either normally or abnormally, it is possible to add a debug package to allow the user to display the contents of any of his variables.

4.2. Minimal Resource Use

Since the Plato system supports simultaneously several hundred students, it is very restricted in the resources it can give to an individual student. The expected limits on resource use were described in section 3.6. In order to assure that the WITS system would work once it had been implemented, its design emphasized minimal resource usage. With this design we hoped to approach as close as possible to these limits, but we knew we could never reach them. Important resources we considered included disk space, disk accesses, ECS requirements, and CPU usage.

In designing the filing lessons, the major resources under consideration were the disk space required to store many programs, the disk accesses required to save or fetch a program, and the ECS needed by the filing lesson and its directory of the disk. These amounts we used of each of these resources could be traded off to have an improvement in one of the others. For example, keeping the directory of programs on disk would reduce

the amcunt of ECS needed, but would increase the number of disk accesses required to manipulate programs. Keeping all programs in a single blcck on disk would reduce the number of disk accesses, but wculd increase the disk space needed. Three important pcints were considered in deciding how tc trade off various resource uses. First, if the filing lesson used too much ECS, at times it might nct be accessible to the student due tc a shortage cf ECS, even if all the student wanted was to save his prcgram and leave. Second, if the directory for the programs was kept in ECS, it would be possible for the disk to be changed, and a system crash tc cause the changes to ECS ccmmon tc be forgotten, resulting in an cut of date directory. Third, the Plato system can regulate the speed at which disk accesses are done, but ECS usage and disk space usage are pre-determined quantities. Thus, if we could reduce our ECS use or disk space use by requiring more disk accesses, it might make WITS mcre usable, if cnly slightly slower.

In the editors and compilers in WITS, the two important resources we had to consider were ECS requirements and CPU usage. Whenever there was a questicn of whether to implement scme language or compiler feature, the usefulness of that feature was traded cff against the resources needed by it, the work required to implement it, and the effort required to maintain it.

4.3. Language Changes

When some language feature appeared too costly to be implemented in WITS, we often felt obligated to provide some substitute feature. The best example of this is the use of READ-WRITE-FORMAT statements in standard Fortran. Due to the complexity of these statements, both in syntax and semantics, much compiler code would have to be written to compile these statements, and the CPU time needed to execute them would slow down interpretation significantly. The language features provided, as an alternative, were format free PRINT and READ statements. In fact, it is becoming standard to use format free I/O statements in introductory programming courses, as evidenced by a standard Fortran programming textbook, "Ten Statement Fortran Plus Fortran IV" [Kennedy,1970]. Thus, we felt no obligation to stick to programming language standards when such standards required significantly more resource use.

One justification for leaving out expensive compiler features is the improved response this allows. The student learns nothing while the compiler is computing; it is only when the compiler or interpreter interacts with the student that the student can learn. Thus, the faster the compiler can respond to input from the student, the faster the student can learn. Learning is especially hindered if the compiler system is so slow as to not even allow the student to enter his program at a reasonable speed.

Another justification for leaving out expensive compiler features is the increased compiler usage this allows. The author benefited in high school from the availability of free computing time, limited only by the availability of terminal time. One of his personal goals in designing the WITS system was to provide a similar system, where a student could write and run as many programs as he desired. The only way he could do this was to make the WITS system as cheap to use as possible.

4.4. Stable System

As authors on the early Plato IV system, we became well aware of what occasional system crashes could do to certain types of data storage methods and structures. As WITS developed, we endeavored to design our data structures so as to be unaffected by system crashes. Of course, we were not always entirely successful, but we did succeed in making some structures less affected by system crashes. For example, the directory in ECS for one of the filing systems was designed so it didn't change when programs were saved or retrieved from disk. Thus, a system crash did not result in an incorrect file directory. Another example would be that if a student enters WITS with a program saved in his student record, the automatic checkpointing of his record to disk while he is running is turned off. That way, if the system crashes while he is running, his signon record with the program in it will not have been overwritten by whatever was in his student bank while he was running.

Chapter 5

The WITS System Now

The first few chapters of this thesis described the users of WITS and the computer system on which it runs. The fourth chapter described the design decisions we made in building it. Together these chapters detail why and how the WITS system was implemented. This chapter describes the resulting system, its appearance, its users, and its operational characteristics.

5.1. Structure

The first BASIC system on Plato was implemented with filing system, editor, compiler, and interpreter all contained in a single lesson. In order to expand the system to include more compilers we had to break it into connected modules, with well defined interfaces. The structure of Tutor allowed the BASIC compiler/interpreter to be placed in one lesson, accessible from a number of editors; and the editor/filing system to be placed in another lesson, with access available to a number of compilers. At the same time, work was in progress on a Fortran compiler, and these changes allowed the existing editor and filing system to be used to test this new compiler.

Currently there are three editors available in the WITS system. The first, called "linedit," is essentially the same editor as was used in our original BASIC system, though it has

been rewritten twice. This editor is line oriented, and looks quite similar to the editor that was available on the PLORTS system. In our implementation, linedit is quite suited for BASIC programs, but may also be used for other languages.

The second editor built for the WITS system, called "cursedit," is a cursor oriented editor, quite similar in appearance to those used in the CAPS system. All editing actions take place at a "cursor" on the screen. The user has available function keys both for positioning the cursor on the screen, and for performing line, word, or character operations on the text of his program. The ability to specify line numbers in cursedit has not been implemented yet, i.e., it is not yet suitable for editing BASIC programs. It has, however, been used by CS105 students for running Fortran programs.

The third editor, called "tutedit," is currently under construction. This editor is similar in appearance to the standard Plato source editor, and is intended for those users already familiar with such an editor. We plan for it to be able to handle programs with and without line numbers, so it may be used for both BASIC and other languages.

All three of these editors will use very much the same internal format for source programs, in order to allow the user to move with his program from one editor in WITS to another without requiring a lot of processing. This allows him to access any of the compilers in the WITS system, to move to any of the other editors, and to use any filing system in WITS.

Also, by standardizing the internal format of user programs, and the interface requirements between lessons, we have made it much easier for additional lessons to be added to the WITS system.

Two filing lessons are currently in use in the WITS system. The first was designed in the fall of 1973 along with the line oriented editor, and was built into that editor. Each program in this filing system is stored in a separate block on disk, thus requiring only a single disk access to either save or fetch the program. The directory for the disk resides in the lesson's common, and indicates for each student signon the block numbers that the student may save programs in. The student references these blocks by way of letters A, B, C, through J, which the lesson uses to index into the list of blocks for that student. This results in a filing system that requires very few disk accesses and a small directory in ECS, and is very stable since the directory does not change when programs are saved or fetched. A large amount of disk space is "wasted," though, since blocks are allocated even if there is no program saved in them, and because only one program is placed in each block.

The second filing lesson in WITS was designed in the fall of 1975, and was built to reduce the amount of disk space needed for saving student programs. Two methods were used to reduce this space. First, programs were packed to allow more than one program per disk block, and second, space was not allocated in the disk file until a program was actually to be saved. This did result in a substantial savings in disk space.

Unfortunately, it also resulted in a very large directory for the disk file in ECS, and a directory that changed when programs were saved or deleted.

Currently accessible from the editors in WITS are working compilers for Fortran and BASIC. The BASIC compiler is essentially the one used in our original BASIC system of 1973. The Fortran compiler is much newer, and is not really complete yet, even though it has enough basic features and is reliable enough that it was used briefly for CS105 students in the fall of 1975. Of course, these are not real "compilers," they have to generate an internal code for the user's program and then interpret that internal form. Thus each of these compilers is built in two parts, the compiler section and the interpreter section, which could be in separate lessons, but are currently together in one lesson.

Two other compilers are currently accessible in the WITS system, but only for testing purposes. One is for an extensible language called "extran," and has been worked on by Bruce Parrello. The other compiler is being constructed by Bruce Copland, and should be able to execute a subset of PL/1 next fall.

5.2. Sophistication

There are several reasons for making the WITS system fairly unsophisticated, as have been mentioned in previous chapters. Our attempt at unsophistication has been very successful. We

provide a very basic interactive compiler system on which students can learn to write programs. No attempt has been made for our lessons to teach the student how to program, but only to let him practice what he has learned elsewhere. With this goal, the best thing we can do to help him learn to program in the real world is to provide a system where he can learn by practice.

5.3. Speed

We have been impressed with the speed at which our compilers and interpreters run; we would not have guessed that Plato could have supported such straight computing as is done in compilers and interpreters. If one compares the turnaround time of many batch systems with the time it takes to compile a program in one of the WITS compilers, one finds that WITS often takes less time. This does not mean that WITS is fast. On the contrary, waiting at a terminal while nothing is happening on the screen can be much more aggravating than turning in a punched deck of cards and going to read a book while the job sits in the computer. Just to have some real statistics, it takes approximately 100 seconds real time to compile a 40 line program in the Fortran compiler at the current Plato CPU limit. Interpreting the internal form of these compiled statements averages 1.5 to 1.9 lines per real time second.

5.4. Resource Use

Besides CPU use, the lessons in the WITS system use other resources, notably ECS space and disk space. Breaking up the WITS system into modules contained in separate lessons has decreased the ECS needed to run a single user, since he is in only one lesson at a time. For a whole class of users, however, the ECS use has only decreased slightly from our first BASIC system. In particular, the BASIC compiler requires 5900 words of ECS, while the Fortran compiler currently uses 8300 words. The line oriented editor with the built in filing system requires 4400 words of ECS, while the cursor editor requires only 3300 words. In addition, each user in any of the WITS lessons will have his own storage area in ECS, whose length will be anywhere from 320 to 1000 words long. The default length of 320 words works quite well for programs up to about 60 lines long using none or a few small arrays.

Disk space is used by the WITS lessons for permanent storage of user programs. The space needed to store each program is about 60 words more than the length of the actual text in his program. Most of these extra words are used to keep a line table, describing the position and length of each of the lines in his program. The rest of these extra words contain a header containing what WITS needs to know about the program. Most of the groups using WITS have allocated enough disk space for one program per student to be saved. This is apparently adequate for most of their needs, especially since the WITS

editors will often be able to save one program in the user's student variables between sessions.

5.5. Stability

As indicated earlier, one of the goals of the design of the WITS system was that it be stable, even under occasional system crashes. We are quite pleased with this aspect of much of our system. The line oriented editor and filing system, as previously mentioned, does not require changes to its directory of disk when programs are saved or deleted from disk. Since the directory is kept in common for easy access, this is very important, otherwise it would be possible for the directory to be updated in ECS, for the changes to be made on disk, and then for the system to crash during the interval before the common is returned to disk.

5.6. More Work Needed

We are quite pleased with the speed and reliability of most of the WITS system, as is probably apparent from the above descriptions, but that doesn't mean that all the problems in the system have been solved. In particular, the stability of a filing system recently added to WITS is somewhat unknown. In trying to reduce the amount of disk space needed to store programs, its directory stored in common in ECS required changing when files were saved and destroyed. Luckily Plato's reliability has improved during the past few years, but even so,

a better algorithm in that filing system should be attempted.

Also, many improvements are possible in those parts of the system that are already being used by students. In particular, a few extra features might be added to the BASIC compiler, such as MATRIX statements. The Fortran compiler can use a lot of work. Many useful features of Fortran have not been implemented yet, such as full array input/output, or DATA and COMMON statements. Also, the error diagnostics of both the Fortran compiler and the BASIC compiler need to be improved, in order to help the student learn where his errors are.

Chapter 6

The History of WITS

As the reader has already learned, WITS has been in use for almost three years now, in one form or another. It started as a BASIC compiler, editor, and filing system contained in a single lesson, capable of running 10 students at once and storing maybe 30 programs. It has grown during that time to the current system, described in the last chapter, which can support any number of students at once and store as many programs as the user can get disk space for. The sections of this chapter describe some of the steps along the way to the current WITS system.

6.1. Changes to Plato Affect WITS

When we began to design WITS, in early 1973, the Plato system was still experimental. It could support 150 users with difficulty, crashes could be expected daily, and strange problems with Tutor lessons could often be traced to errors in the run-time interpretation routines. Many useful features of Tutor were still to be developed, but the form of the language, and most commands, remain today as they were then. The major language changes that affected the development of the WITS system are described below. Some of these descriptions are rather technical, and might only be understood after some

experience on Plato.

force micro / altfont alt

Plato terminals have over 120 built in characters, with internal codes of six, 12, or even 18 bits used to reference them. In order to reduce the complexity of our original BASIC compiler, we used six bit internal codes for all characters. This shortened the amount of space needed to store the student's program, and more importantly, it simplified the implementation of the editors and compilers. This required the editor for BASIC programs to collect key presses and convert them to the desired internal codes. This naturally required both Tutor code and CPU time.

In 1973, some time after our first BASIC system was written, the Plato system announced a new command, -force micro-. Normally a micro table is used to substitute a series of characters for a single key press, such as substituting the equation of an ellipse for the student's press of MICRO-e. The -force micro- command directs Plato to translate all key presses through the micro table, even if the user has not hit MICRO. The micro table in the WITS system translates those keys in our character set with six bit internal codes into themselves (about 50 of these). Those keys for characters with 12 or 18 bit codes get translated into unused six bit codes. For example, the single quote gets mapped into the code 53, which is normally used for the Tutor assignment arrow.

Since it would be a bit disconcerting for the student to see an assignment arrow when he typed a single quote, we use an alternate character set to change the display representation of our six bit internal codes. Normally charsets are used to display often plotted characters or figures without drawing them dot by dot each time, such as Russian characters in a Russian language lesson. Charsets are loaded into the read/write character memory of the terminal once and then can be referenced as fast as references to the built in, read only, character set. The charset that WITS uses looks like the built in character set, except it is rearranged a little and has fewer characters.

Side affects of having all input come through a micro table and all output being shown through a charset are nice things like having Shift-o delete a character as on a teletype (the micro table translates Shift-o into a single press of ERASE), and displaying capital letters for lower case internal codes (just making the designs in the charset look like upper case letters). It should be noted that numbers and operators in expressions get mapped into themselves. This allows the use of standard Plato routines at execution time to process input expressions and to display numerical results.

Time Slice Changes

In early 1973, the programmer of a Tutor lesson indicated, by means of a -break- command, where Plato should check to see if he was nearing the end of a timeslice. If he was, Plato

would interrupt him at that point. When he returned from an interrupt, all his *nc* variables would be zero, so before a -break-, he would generally -unload- his common and storage from *nc* variables into ECS. After the -break- command, he would -load- his *nc* variables from ECS again. During a timeslice, he could use *nc* variables that were not loaded from ECS, and their values would remain intact for the duration of the timeslice. These variables provided "temporary variables," which could be used much like the local variables available in many other programming languages.

In August, 1973, -break- commands became obsolete, with the Tutor interpreter determining when a timeslice interrupt, or -autobreak-, should occur. Since an -autobreak- could happen between any two commands, temporary variables could now be depended on only for the duration of a single command, and even that may change in the future. Before this time we had been using temporary variables extensively, so removing -break- commands required many hours of our time to get our lessons working again.

The commands -load- and -unload- were only usable when one knew where a -break- would occur, so in addition to making temporary variables obsolete, removing -break- commands also required a change in the manner of referencing common and storage. Two new commands, -comload- and -stoload-, were provided to automatically load and unload portions of common and storage every timeslice.

The language resulting after the implementation of -autobreak- was cleaner and more powerful than the previous version of Tutor. However, some 40 hours of rewriting were required to get the BASIC filing system, editor, and compiler, to work with -autobreak-. Much of this could have been avoided if we had known a year and a half ahead of time what changes were planned for Tutor. However, probably no one knew at that time what was going to be done.

6.2. Resource Shortages

Periodically during the history of Plato, various resources have been in short supply compared to the demand for them. Below are descriptions of each of the resource shortages the WITS system has survived, and how each affected the development of WITS.

Disk Space Crunch

As with many other computer systems, Plato has to be careful in allocating its disk space to users. At one time disk space was given to users freely, but it was soon learned that such policies wasted disk space. But, even with careful allocation, sometimes new disk drives didn't arrive when expected, or the drives would arrive, at which point it was found that the necessary disk controller had not even been ordered. Sometimes Plato was short of disk space for months at a time. Only rarely, however, did the WITS project slow its

development due to the lack of space in which to author lessons, but thought was given continually to the amount of space data structures would take on disk, and how many students could run with a particular disk setup. The original BASIC system, which required a copy of the entire filing system, editor, and compiler for every group of users, wasted a lot of disk space compared to later versions of WITS.

ECS Crunch

Occasionally, during the three years that the author has been running compilers, the number of users on Plato has grown to where all the available ECS has been in use. When trying to run classes in instructional lessons, this results in students sometimes being unable to sign on to the system, or unable to move from one lesson to another. The original BASIC system resided in a single lesson, and since any number of users can share the ECS copy of a lesson, that system rarely had trouble running classes. Running just one student during unscheduled time was often impossible, however, since he was unable to obtain a large amount of non-shared ECS.

When WITS was built, the compilers resided in lessons separate from the lesson with the editor and filing system. With this setup, it was possible for all students to sit in the editor with the compiler unaccessible due to lack of ECS. The solution, once Plato allocated ECS to physical sites, was to reserve a classroom for a whole class of students running in the

compiler, and to ask other users in the classroom to leave whenever ECS became scarce. Running a single student became easier with this setup since he was only charged for the ECS of the lesson he was using at the time, either the compiler or the editor, both of which were smaller than the original BASIC system.

The current WITS design assumes classroom use where both the editor and compiler can be continuously shared between many users. When modifications are made to WITS, attempts are made to shorten the total ECS it uses. Also, it is desireable that if the ECS situation becomes too tight to run, the student should be able to get to the filing system and save his program. Uses of WITS for other than whole classes at once will have to be restricted to times when ECS is plentiful.

CPU Crunch

As the number of Plato users has grown, the CPU time available to each has diminished. To guarantee a certain amount of processing time for each user, the Plato system programmers have added software checks to control CPU allocation. Even so, the Plato system has occasionally felt sluggish to experienced users. The worst such time was for about two weeks during the 1975 Spring semester, when Plato was running over 400 users regularly. Improvements to the Plato interpreter during the following summer allowed it to handle peak loads of well over 500 users during the 1975 Fall semester with hardly any

ncticable delay. Since CPU use is a critical resource, the WITS designers have attempted to keep processing time to a minimum without increasing use of other resources (such as ECS). Their techniques for this are described below.

The WITS system consists of several interconnected lessons. Since a user can go from any lesson to any other lesson, and will often move from an editor to a compiler and back, it is important to keep the processing time at such interfaces small. This is done by standardizing the internal format of the user's program in all the lessons. Additionally, it is important that no compiler destroy information that an editor would have to use CPU time to recompute. At various times during the development of WITS, some lessons violated the above two rules. The BASIC compiler has traditionally destroyed information the editors want, just to save 15 words of ECS; and a new editor used a different format for programs in order to reduce its own processing, but more processing was required to access a compiler than was saved. Both of these violations have been corrected, with a savings in processing time, and only a minor increase in ECS usage.

The CAPS compiler system has a much different structure than WITS has. CAPS compilers, rather than being hard coded Tutor, are coded in an assembler-like language, and are interpreted by a common driver, written in Tutor. Since the driver is shared between all the compilers, it loses some of the speed it might have if it had language dependent optimizations

built in. To save processing time, the WITS compilers are coded directly in Tutor.

A third way of keeping CPU use by the WITS compilers and interpreters low was to utilize Tutor commands and data structures that were known to be fast. Such things included using -find- commands where appropriate, referencing full words rather than segments of words when possible, and referencing information directly rather than through links and pointers.

Probably the greatest effect on CPU time and ECS consumption was not from any one thing done during the design of the compilers, but rather on the method used in building the WITS system. All parts of the system were designed by one or more people in discussion with other authors, but only one person implemented each part. The system was thus broken down into parts that could be thoroughly understood by a single person, allowing him to recognize optimizations when he saw them. Often, after a portion of the system was completed, it was examined by the other WITS designer to make sure it was understandable to both. The other designer would occasionally suggest improvements, and both were then competent to correct bugs that appeared in the lesson.

Chapter 7

Conclusion

Previous chapters have described who the WITS compiler system was built for, how it was built, and what its current appearance is. Mention has been made several times to CAPS, an acronym for the "Computer Assisted Programming System" that has been developed on Plato by the Computer Science Department at the University of Illinois. This chapter compares and contrasts these two student compiler systems.

The CAPS programming system was designed to fill part of the role of a consultant in helping the student write his program. As the student enters his program in one of the CAPS editors/compilers, it is checked, character by character and token by token, for correct syntax. Immediately upon receipt of an invalid token or syntactic construct, the compiler gives an error message to the student, and the editor will not let him proceed until he has corrected his mistake. If the student does not understand the first error message, which generally is quite short, he can press HELP, and receive a more detailed explanation. If the student still does not know how to correct his program, he can direct the compiler to give him a series of changes, each one of which would make his program correct syntactically. The student can make one of these changes, and continue entering his program. This is a very impressive

operation, examples of which can be seen in [Tindall,1975].

At program execution time, the CAPS interpreters keep trace information at every step of interpretation. If the student receives an execution error during interpretation, this allows the diagnostic routines to actually reverse execute the student's program, showing him each step of the execution, until he can determine where the error in his program was. This process is further described in [Davis,1975].

In the area of compile time and execution time diagnostice, the WITS system is severely lacking. At present, the student is essentially told that the compiler can not understand some line. A hint might be given as to what the compiler is expecting next, but the student is not even told where in the line the compiler was processing. Essentially, the student is expected to already know the syntax for his programming language, and the WITS compilers allow him to write programs in that language. When the student receives a compiler error message, he may have to look up in a reference book the syntax for the particular statement he wants to use, and then follow that syntax exactly.

In the area of execution speed and resource use, the WITS system comes out better than the CAPS system. The editor/compiler in the CAPS system must be able to compile forward performing syntax checking, and uncompile backward as the studert moves his cursor back to change a previous statement. At execution time, the CAPS interpreters must keep trace information in order to allow their reverse execution

routines tc wcrk. This naturally requires more CPU time to collect this data, and more ECS space to keep the data for later use. An analysis of where CPU time was being spent during ccmpilaticn in the CAPS system was performed last year, and is described in [White,1975].

In summary, the Platc system has two student compiler systems currently being developed. Each system is designed differently and has different operaticnal characteristics. The CAPS ccmpiler system is designed to watch the student as he enters his prcgrams, and help him whenever he has trouble. The WITS ccmpiler system, described in this thesis, is designed for speed and minimum resource use, in crder that the student can learn programming by receiving lcts of practice. Because of its scmewhat simpler design, the WITS system has been slightly more reliable while in the construction stages, though this should disappear as the CAPS system is polished up. Both of these systems need mcre wcrk, the CAPS system to speed it up, the WITS system to improve its error diagnostics.

List of References

[Alpert,1970] Alpert, D. and D.L. Bitzer, "Advances in Computer Based Education," Science, Vol. 167, March 20, 1970, pp. 1582-90.

[Bitzer,1973] Bitzer, D.L., B.A. Sherwood and P.J. Tenczar, "Computer-Based Science Education," CERL Report X-37, Computer-based Education Research Laboratory, University of Illinois, Urbana, Illinois, May 1973.

[Davis,1975] Davis, A., "An Interactive Analysis System for Execution-time Errors," Ph.D. Thesis, Department of Computer Science Report # UIUCDCS-R-75-695, University of Illinois, Urbana, Illinois, January 1975.

[Gear,1969] Gear, C., A. Whaley and N. Weidenhofer, "The University of Illinois PLORTS System--Provisional User's Manual," Department of Computer Science File No. 793, University of Illinois, Urbana, Illinois, March 1969.

[Hyatt,1972] Hyatt, G., D. Eades and P.J. Tenczar, "Computer-Based Education in Biology," BioScience, 22-7, July 1972, pp. 401-09.

[IBM] CALL/360-OS BASIC Language Reference Manual, GH20-0699, copyright 1970, 1971 by International Business Machine Corporation.

[Kemeny,1967] Kemeny, J.G. and T.E. Kurtz, BASIC Programming, John Wiley and Sons, New York, 1967.

[Kennedy,1970] Kennedy, M. and M.B. Solomon, Ten Statement Fortran Plus Fortran IV, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

[Lower,1976] Lower, S.K., "Authoring Languages and the Evolution of CAI," Simon Fraser University, Burnaby B.C. Canada, February 1976.

[Nievergelt,1974] Nievergelt, J., E.M. Reingold and T.R. Wilcox, "The Automation of Introductory Computer Science Courses," A. Gunther, et al. (editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[Schreiner,1972] Schreiner, A.T., Computer Calculus, Stipes Publishing Co., Champaign, Illinois, 1972.

[Sherwood,1974] Sherwood, B.A., The TUTOR Language, Computer-based Education Research Laboratory and Department of Physics, University of Illinois, Urbana, Illinois, 1974.

[Stifle,1972] Stifle, J., "The Plato IV Architecture," CERL Report X-20, Computer-based Education Research Laboratory, University of Illinois, Urbana, Illinois, May 1972.

[Stifle,1973] Stifle, J., "The Plato IV Student Terminal," CERL Report X-15, Computer-based Education Research Laboratory, University of Illinois, Urbana, Illinois, June 1973.

[Tenczar,1974] Tenczar, P.J. and B.A. Sherwood, "limits," Plato System Feature Notes, Urbana Plato System, Urbana, Illinois, September 24, 1974.

[Thornton,1970] Thornton, J.E., Design of a Computer, Scott, Foresman Co., Glenview, Illinois, 1970.

[Tindall,1975] Tindall, M.H., "An Interactive Compile-Time Diagnostic System," Ph.D thesis, Department of Computer Science Report # UIUCDCS-R-75-748, University of Illinois, Urbana, Illinois, October 1975.

[Tucker,1971] Tucker, P., "A Large Scale Computer Terminal Output Controller," CERL Report X-27, Computer-based Education Research Laboratory, University of Illinois, Urbana, Illinois, June 1971.

[White,1975] White, L.A., "CAPS Compiler CPU use Report," Department of Computer Science, University of Illinois, Urbana, Illinois, December 1975.

[Wilcox,1975] Wilcox, T.R., "An Interactive Table Driven Diagnostic Editor for High Level Programming Languages," Department of Computer Science, University of Illinois, Urbana, Illinois, 1975.

[Wilcox,1976] Wilcox, T.R., A.M. Davis and M.H. Tindall, "The Design and Implementation of a Table Driven, Interactive Diagnostic Programmin System," to be published in the CACM.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-819	2.	3. Recipient's Accession No.

4. Title and Subtitle

The Design of WITS: A Student Compiler System on PLATO IV

5. Report Date: July 1976

6.

7. Author(s): Lawrence Allen White

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

13. Type of Report & Period Covered: M. S. Thesis

14.

15. Supplementary Notes

16. Abstracts

This thesis discusses the design of an interactive student compiler system called WITS, an acronym for "White's Interactive Timesharing System." The WITS system currently contains two editors, two filing lessons, and two compilers. Work is in progress on creating additional editors and compilers, in order to better meet the needs of its users. These compilers have been implemented on the PLATO IV computer system, and are being used by students. Included in this thesis is a description of the PLATO IV system and how it affected the development of WITS.

17. Key Words and Document Analysis. 17a. Descriptors

interactive student compilers
PLATO IV

7b. Identifiers/Open-Ended Terms

7c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 54
	20. Security Class (This Page UNCLASSIFIED	22. Price